# 1. Introduction

## 1.1. Project Description

The goal of this project is to provide Natural Language Processing (NLP) services in the form of APIs. NLP is one of the major tasks in Machine Learning (ML). Whenever someone finds themselves wanting a computer program that can carry out NLP tasks for them, they would have to go through very time consuming and difficult processes of data collection, data cleaning, finding people skilled enough to do something with that data, build an effective model that can ultimately be used to do the NLP tasks. Instead, an individual or an organization can make use of the APIs provided as a result of this project do these tasks. All they have to do is make API calls with their inputs and the API returns a JSON object with predictions. This is going to save the users months of time and effort. Users can make calls using any language that supports fetching JSON data from the browser. This gives them incredible flexibility while building their applications. It also allows them to focus more of their manpower on their main products. There is always a rising need for machines more powerful, intelligent, and faster than us to take over the tedious and repetitive tasks that we just don’t want to do. ML has provided solutions exactly to these tasks over the years. If someone is conducting a survey on a certain topic, they would not want to spend hours looking at tweets and determining the general sentiment about that topic. They would rather feed that information to a program and let it tell them what kind of sentiment the tweets are showing. The tricky part is getting that program to work properly and accurately. The project uses three different models, trained for three different tasks (Sentiment Analysis, Category Prediction, Spam Detection). All of them averaging an accuracy of over 94%. So, accuracy isn’t a problem anymore. The project was built on node.js using a tensorflow.js integration. This means most of the tasks are done on the browser itself, making it faster and requiring very less computational power.

### 1.1.1. Problem Definition

Text classification is a very important task in supervised machine learning. A piece of text is assigned to one or more classes or categories. This can be done manually or with the help of powerful machine learning algorithms. The problem with doing this manually is that it takes up a lot of time and resources. Let’s say you own a blogging website or a news website. Every article that is being posted has to be classified and put into a category. Making people read these articles manually is both time consuming and expensive. It would be easier if the computer itself classified these articles, as soon as they are posted. This is where the need for Natural Language Processing arises. Natural Language Processing or NLP, is a Machine Learning (ML) task that is used to train an ML model to recognize text data and get meaningful insights from it. This means that a trained ML model will be able to go through some text data and give us some context on it. So, if you pass an article as input, this model will be able to tell you where it belongs. NLP can also be used to do other interesting tasks such as Sentiment Analysis. This means that a model will be able to tell if some text data is positive, negative, or neutral about any topic that is in discussion. Our phones and email accounts are bombarded with spam every day. The only way to filter out the spam is by either making users flag the messages as spam manually or filter out the messages at the server end itself using an effective program. Context Analyzer provides solutions to all three of these tasks.

### 1.1.2. Purpose

The problem of classifying text can be done by the organizations or the users themselves. To do that they would have to start collecting data, hire skilled ML engineers, spend a lot of money and time building out an effective and accurate model. Then some more time has to be spent tuning the model to perform better on all kinds of data. All of this can be avoided when companies just use the APIs provided by this project. The purpose of this project was to automate cartain NLP tasks and also provide them as services through APIs. Integrating APIs is much cheaper and faster than building an entire team to carry out these tasks.

### 1.1.3. Scope

The main users of this project will be those looking to integrate NLP tasks into their application with very little human effort or resources and get very efficient results. The web application also provides a UI which can be used to carry out these tasks for one time users. The sentiment analysis API can be used for predicting sentiment of data from social media platforms, reviews on products, etc.. Category prediction is a multi-class classification task that can be used on news articles or blogs to classify them into different classes like politics, entertainment, sports, health, etc.. The spam detection API allows users to determine whether a message (any form of message; like SMS, email etc.) is spam. Initially I identified these three tasks as the major ones. The application is built in a way that it is always easy to add new features or build new APIs and add them.

### 1.1.4. Proposed Solution

The application consists of three different models all built using a Convolutional Neural Network, or CNN. The three models are trained on; a news dataset, the IMDB reviews dataset, and an SMS Spam dataset. The news dataset is going to be used to train the model for the multi class classification task. The IMDB reviews dataset is going to be used to train the model for sentiment analysis. The SMS spam dataset is used to train the model for spam detection. These models, once put into production will be able to do these classification tasks in mere seconds. This will also be cheaper and more effective.

To provide APIs, Node.js will be used. This has very good integration for tensorflow.js. Which means all ML tasks can be done on the browser itself. Node.js is also an amazing javascript runtime that can be used to build highly efficient endpoints. The web application can also be used as an external tool for the classification tasks. Once the dataset is loaded, the preprocessing starts.

First I use the stop words function to remove the prepositions like “the, of, he, she etc”. It also simplifies the words, for example if there are multiple words like “doing, did, done” it will be converted to “do”. Then the tokenizer is used to turn the text into sequence of numbers. The neural network takes only numbers as input. Once the preprocessing is done, the model is created with several layers. The output layer returns a value denoting which class the text belongs to. Once the model is trained and saved, tensorflow js is used to load it. After this, everything is done on the browser. The predictions are made at the node endpoints.

# 2. Literature Survey

## 2.1. Background Study

A lot of research goes into NLP almost every day. Providing APIs for NLP tasks is a bit of a complex task. One such way to build NLP APIs is using a robust back-end technology like Node.js and making use of its brilliant integration with Tensorflow.js. Tensorflow.js allows us to interact with ML models directly from the browser. This makes it currently one of the best libraries for building NLP APIs. Tensorflow.js, for a while was running only on experimental Node.js. Recently it was ported to latest stable build of Node.js. It shows that this is a good way to move forward.

### 2.1.1. Existing Systems

Google Cloud Natural Language provides NLP APIs. They also make use of tensorflow.js to provide the service. OpenNLP and Stanford NLP provide NLP libraries that can be integrated with supported languages. TextRazor provides NLP APIs but they do not use tensorflow.js. All these systems have certain drawbacks and gaps that can be filled with Context Analyzer API. The drawbacks and the solutions for them are mentioned in a future section.

### 2.1.2. Related Work

**Title**: Recurrent Neural Network for Text Classification with Multi-Task Learning

**Authors**: Pengfei Liu, Xipeng Qiu, Xuanjing Huang

**Summary**: Neural network based methods have obtained great progress on a variety of natural language processing tasks. However, in most previous works, the models are learned based on single-task supervised objectives, which often suffer from insufficient training data. In this paper, we use the multitask learning framework to jointly learn across multiple related tasks. Based on recurrent neural network, we propose three different mechanisms of sharing information to model text with task-specific and shared layers. The entire network is trained jointly on all these tasks. Experiments on four benchmark text classification tasks show that our proposed models can improve the performance of a task with the help of other related tasks.

#### **Title**: Long Short-Term Memory

#### **Authors**: Sepp Hochreiter, Jurgen Schmidhuber

**Summary**: Learning to store information over extended time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow. We briefly review Hochreiter's 1991 analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called "Long Short-Term Memory" (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error flow through "constant error carrousels" within special units. Multiplicative gate units learn to open and close access to the constant error flow. LSTM is local in space and time; its computational complexity per time step and weight is O(1). Our experiments with artificial data involve local, distributed, real-valued, and noisy pattern representations. In comparisons with RTRL, BPTT, Recurrent Cascade-Correlation, Elman nets, and Neural Sequence Chunking, LSTM leads to many more successful runs, and learns much faster. LSTM also solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms.

### 2.1.3. Drawbacks of Existing Systems

Unlike Google Cloud Natural Language, this project allows users to customize the API results however they want. The response from the API is a JSON object, which means users can fetch any data they want from the output easily. Also, making API calls is easier than ever as all it requires is the input to be appended to the URL. (for ex: “localhost:3000/api/sentiment?predict=your+input”). A simple GET request would give you the predictions in a JSON response. While Open NLP and Stanford NLP provide good libraries, it wont be easy for every user to download the library and write code to make use of it. It is much easier to just make API calls and get output almost instantly. The major problem that Context Analyzer is solving is the ease of use and integration. As stated above, making API calls is very easy and simple for everyone to understand. The JSON response is also very easy to decode and every part of the response can be fetched individually based on the user’s requirements.

## 2.2. Feasibility Study

### 2.2.1. Technical Feasibility

The application runs completely on the browser. This does not require any additional installations or downloads. It is extremely easy to add new APIs or features when built. The back-end uses all the es6 features of Node.js which means it is highly responsive and fast. Obviously, the development part is hidden completely from the user. All the user will see is a responsive UI.

### 2.2.2. Economic Feasibility

The development of this web application barely costs anything. It can be built with computers that have a decent amount of computational power. I decided to use the CPU version of tensorflow and tensorflow.js. This means it can be built using devices without a GPU. All the tools and technologies used were open-source, so, no licensing was required.

### 2.2.3. Operational Feasibility

Operating the web application is also very easy. The web application is built completely on the browser, so it will be fast and responsive. The application also contains documentation for the APIs. Every new user can go through it and find out how to use the APIs. One time users can also make use of the UI provided for each of the APIs. This takes in the user input and returns predictions. This is effective to show as a demo for the APIs and gives the users an idea of how quickly the results are given.

**2.3. Tools and Technologies**

**2.3.1. Tensorflow / tfjs-node**

**Tensorflow** is an end-to-end open-source platform for ML. It has a wide range of tools and libraries, along with community resources that help us build state-of-the-art ML powered applications. In the context of this project, tensorflow was used to build and train the models. **Tensorflow.js** is an ML library for JavaScript. It allows us to develop ML models in JavaScript and use ML directly in the browser or in **Node.js.** This application made the most use of tensorflow and tfjs-node as it was the core component in building the APIs and models.

**2.3.2. Node.js**

**Node.js** is an asynchronous event-driven JavaScript runtime. **Node.js** allows us to build scalable network applications. **Node.js** served as the main back-end for this project for its **REST API** and **asynchronous** capabilities. A lot of asynchronous code is written to support the loading of models and making predictions. **Node.js** powers all of these tasks effectively.

**2.3.3. Python 3**

**Python** is a powerful and fast programming language. Python is user-friendly and easy to learn. Python basically runs on any platform and is open-source. Python was largely used as the platform for building and training the models. Tensorflow integrated with Python is the most powerful tool for ML. Python’s endless libraries facilitate users to perform almost any task.

**2.3.4. Postman**

**Postman** is an API development tool that helps users test and build powerful APIs. From headers to authentication tokens to simple JSON values, Postman supports all these features. It was largely used in the development part of the web application. All the routes and requests were simulated, monitored, and verified using Postman. Only then would they be integrated with the front-end.

**2.3.5. Handlebars.js / hbs**

**Handlebars.js** is used to build semantic templates. Handlebars is compatible with **Moustache** templates as well. Handlebars templates are compiled into JavaScript functions. These templates allow for reusability and dynamic rendering of content. **Hbs** is an express.js view engine for **Handlebars.js.** Hbs can be used with express.js to seamlessly write handlebars templates and integrate them with the HTML code. Hbs supports Partials and Views. Views are your main HTML pages and Partials support rendering of partial content on the views. This makes for building an effective, dynamic, and lightweight front-end.

# 3. Hardware and Software Requirements

## 3.1. Hardware Requirements

* **Development Environment:**
  + **HDD:** 4GB
  + **OS:** Ubuntu 18.04 LTS, Windows 10, Mac OSx
  + **Processor:** Intel i5 Gen 8
  + **RAM:** 8GB

## 3.2. Software Requirements

* **Server Side:** Node.js 13.x, Express 4.x, tfjs-node 1.5.x
* **Client-Side:** Handlebars 4.x, HTML5, CSS3
* **Environment:** Ubuntu 18.04, Windows 10, Node, Python runtime environment
* **Code Editor / Ide:** VS Code, Jupyter Lab

# 4. Software Requirement Specification

A software requirement specification gives a detailed description of a software system along with its functional and non-functional requirements.

## 4.1. Users

Users are the people who interact with the application. They might be making use of the API or using the web UI to get predictions.

**API User** is the user who uses the API to get predictions. They make API calls from their own application and make use of the response.

**One-time User** is the user who uses the web UI of the application to get predictions for their own input.

## 4.2. Functional Requirements

### 4.2.1. Data Collection

* Deciding what kind of data is required for building the models.
* Downloading pre-collected data from Kaggle, and other sources.

### 4.2.2. Data Preparation and Pre-processing

* Building a dictionary of words from the dataset. This step is important in the pre-processing section of the project as the dictionary is used to get more refined and accurate predictions. This dictionary has the mapping between normalized words and their integer IDs. The gensim.corpora.Dictionary class is used.
* Removing unnecessary fields from the dataset. There might be some unnecessary fields like ‘id’, ‘date’, etc., these should be removed if not necessary for building the model.
* The text data is converted into a list of tokens. The neural network takes only text data as input. The tokenizer() function is used for this. This leaves us with a tokenized list as our ‘X’ and the classes as our ‘Y’. This can now be used as inputs for the neural network.

### 4.2.3. Train the Model

* This is where the pre-processed data is used to effectively train the model.
* Used keras sequential model with 6 different layers.
* The LSTM layer is what helps the model remember context. This allows for accurate predictions.

### 4.2.4. Evaluate the model

* Evaluating the models using some metrics. This can be a classification report or f1-score or any metric of the appropriate choice.
* This gives us a good idea of how the model is performing and lets us know what improvements are required.

### 4.2.5. Make Predictions

* This is when the trained model is used to make predictions.
* The ‘test set’ as its often called, is passed as input to the model and it returns a list of predictions for each row of the test set.
* This gives us a good estimate of how the model will perform.

### 4.2.6. Server side processing

* The user input from the web UI or the API has to be pre-processed before predicting the class. This is done at the server using tfjs-node.
* The predicted output from the model is turned into JSON and sent as response.

## 4.3. Non – Functional Requirements

### 4.3.1. Scaling

Building models for more NLP tasks, and set up APIs for these tasks.

**4.3.2. Performance**

Increase accuracy of predictions. Increase capacity to handle a large number of API calls at the same time.

**4.3.3. Availability**

Ensure that the web-app is always functional and available to use for the users. The server handling the API requests must always be up. Any sort of downtime will cause inconvenience to users.

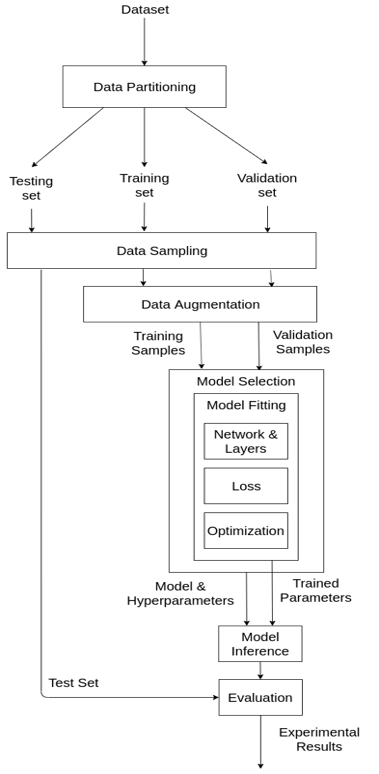
**4.3.4. Maintenance**

Updates and maintenance of the web-app must be easy. Any updates in the technologies used should be easy to integrate with the previous versions.

# 5. System Design

## 5.1. Flow Diagram

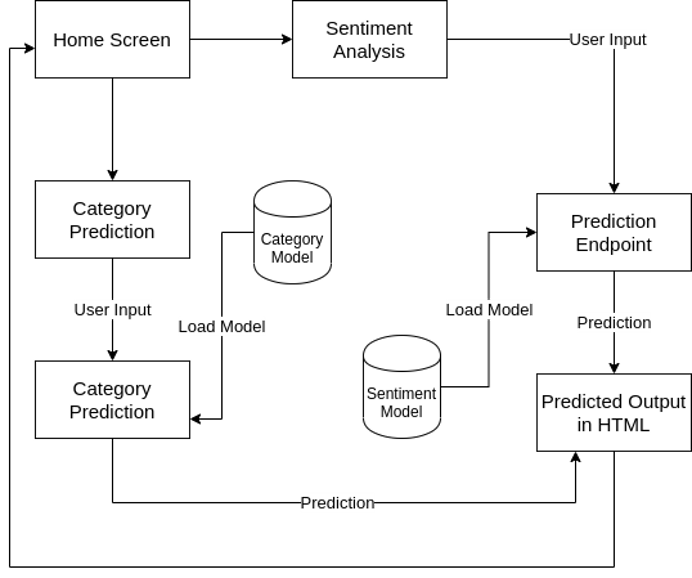
**5.1.1. ML view**



**Figure 5.1.1.a – Flow diagram, ML view**

The flow diagram for the ML view deals only with the ML processes of the project. It shows how data flows between the different modules of the ML model building process.

**5.1.2. Web-application view**



**Figure 5.1.2.a – Flow diagram, Web-application view**

The flow diagram for the web-application view deals with how the different processes communicate in the web-application from server side to client side, and also how the server communicates with the models saved in memory.

## 5.2. Detailed Methodology

**5.2.1. Text Pre-processing**

The datasets contain text data that need to be converted into tensors in order to provide them as input for the CNN. The neural network does not take in raw text data as input, so we need to convert this text data into a sequence of numbers (or a tensor) which is the appropriate input type. But first, a dictionary of words from the dataset is create. Its significance is explained later. This dictionary is built using the **gensim.corpora.Dictionary** class. This contains a dictionary of words with their integer values as keys. This dictionary is of paramount importance as it plays a very important role in increasing the accuracy of the model. The dictionary is then used to convert the dataset into numerical inputs by using the **token2id** method available from **gensim.corpora**. This method converts the dataset into key-value pairs by giving each word in the dataset the integer value from the same word in the dictionary. This process is called **tokenization** and the numbers are called **tokens**. These tokens are then converted into a 1D list which consists only of tokens. The code and outputs from these processes are available in the Implementation section.

The user input for predictions are also pre-processed in the same way. The same dictionary is used. If the dictionary isn’t used the user inputs would be saved as [0,1,2,3,4] and this would lead to very inaccurate predictions. When the dictionary is used, the code checks if the words from the user input exists in the dictionary, and then the appropriate token is assigned. This makes the model give us better predictions.

**5.2.2. Building the model**

1. **Convolutional Neural Networks**

Convolutional Neural Networks, or CNNs, are specialized neural networks that take in input as a 2D matrix. In the case of this project, the input, that is a list of tokens is converted into a series of convoluted matrices. Each row of the matrix corresponds to a token which may be a word or a character. Each row is a vector that represents a word. In NLP the filters in CNNs slide over sentence matrices, a few words at a time.

1. **The layers of the model**

All three models used in this project are similar but with slight differences. What all 3 models have in common is that they all have the same layers but the parameters differ to make sure the datasets were used to train the model with maximum efficiency.

* 1. **Embedding**

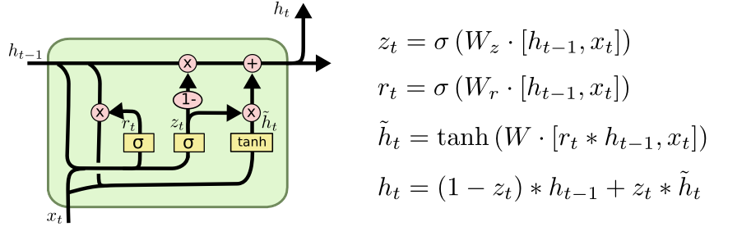
The embedding layer can only be used as the first layer of the model. The embedding layer takes 2 main parameters. The input dimension and the output dimension. The input dimension is an integer that specifies the size of the vocabulary. The output dimension is an integer that specifies what shape the output of the embedding layer must be. This means that we can use the embedding layer to convert higher dimensional data into lower dimensional vector space.

In this project it is implemented like this:

*model.add(Embedding(len(dictionary), embed\_size)*

* 1. **LSTM**

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word of the sentence “the cars are on the *road*”, we don’t need any further context – it’s pretty obvious the next word is going to be “road”. In such cases where the gap between relevant information and the place that it’s needed is small, RNNs can learn to use the past information. But, there are also cases where we need more context. Consider trying to predict the last word in the sentence – “I grew up in Spain. I speak fluent *Spanish*”. Recent information suggests that the next word is probably the name of the language. But if we want to find out which language, we need the context of “Spain” from further back. Hence, it is entirely possible for the gap between information and the place where it is needed to become very large. As that gap grows, RNNs become unable to connect the information. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.



**Figure 5.2.2.a - LSTM**

1. **Model architecture**

The following diagrams show the architecture of the models. They consist of all the layers in the model, showing input and output shapes for each layer. The architecture diagrams for all 3 models; sentiment analysis, multi-class classification, spam detection are shown below.

****

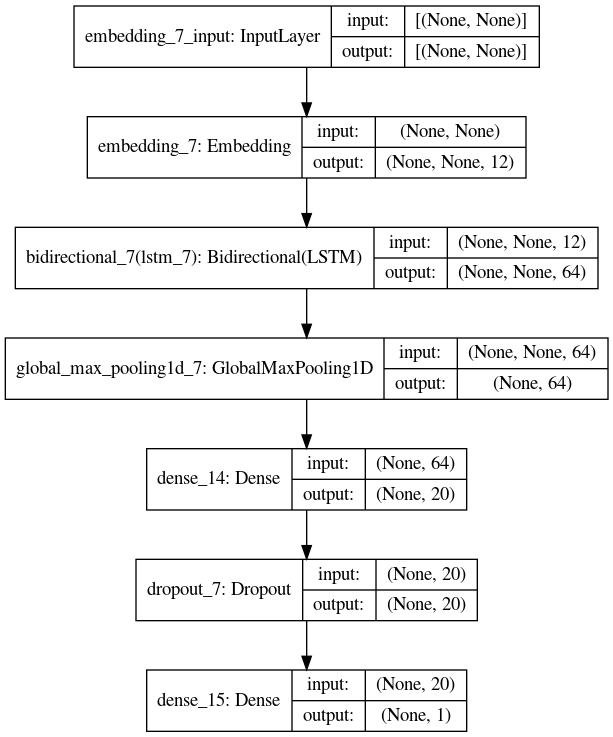
**Figure 5.2.2.b – Architecture of Sentiment Analysis model**

The above diagram shows the layers and input and output shapes of each layer for the sentiment analysis model.



**Figure 5.2.2.c – Architecture of Multi-class Classification model**

The above diagram shows the layers and input and output shapes of each layer for the multi-class classification model.



**Figure 5.2.2.d – Architecture of Spam Detection model**

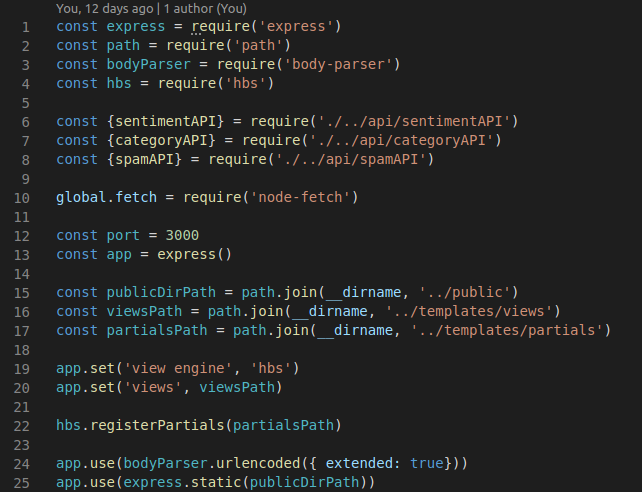
The above diagram shows the layers and input and output shapes of each layer for the spam detection model.

# 6. Implementation

## 6.1. Setting up routes for the web application

### 6.1.1 Sample Source Code

But first, I set up the prerequisites for the server to run. That includes, installing express.js and importing it. Also importing hbs, body-parser, path. Here I also set up the view engine as hbs, this tells the server to use handlebars as the view engine and it serves up hbs files at the client side from the folder specified.



**Figure 6.1.1.a – server.js set-up**

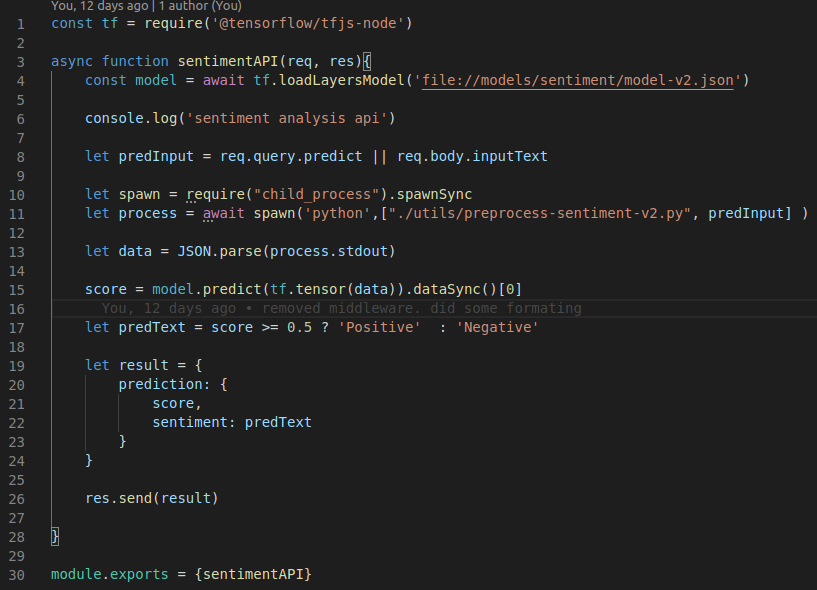
After this, the routes can be written. Most of the GET requests render a hbs file. For example, a GET request made to ‘/sentiment-analysis’ would render a hbs file for the sentiment analysis task.

The routes look like this.



**Figure 6.1.1.a – server.js routes**

When the API calls happen, a function runs to load the model and make predictions and send those predictions as response.

The code for sentiment analysis API.

**Figure 6.1.1.c – sentimentAPI.js**

The other APIs are also setup in a similar way. The asynchronous function setup makes sure that the model is loaded properly. The input is preprocessed and the model is used to predict the processed input. The predictions are sent as a JSON response.

All of this handles the server side of the application. But the most important part of the application is the models. Without them none of this can be used in any meaningful way. So, here is how the models are generated.

The pre-processing is pretty similar for all of the models. Only the size of the input differs as each dataset is different. The code for pre-processing looks something like this.

Code from sentiment.ipynb

MAX\_SEQUENCE\_LEN = 130

UNK = 'UNK'

PAD = 'PAD'

**def** text\_to\_id\_list(text, dictionary):

**return** [dictionary.token2id.get(tok, dictionary.token2id.get(UNK))

**for** tok **in** text\_to\_tokens(text)]

**def** texts\_to\_input(texts, dictionary):

**return** sequence.pad\_sequences(

list(map(**lambda** x: text\_to\_id\_list(x, dictionary), texts)), maxlen=MAX\_SEQUENCE\_LEN,

padding='post', truncating='post', value=dictionary.token2id.get(PAD))

**def** text\_to\_tokens(text):

**return** [tok.text.lower() **for** tok **in** nlp.tokenizer(text)

**if** **not** (tok.is\_punct **or** tok.is\_quote)]

**def** build\_dictionary(texts):

d = Dictionary(text\_to\_tokens(t)**for** t **in** texts)

d.filter\_extremes(no\_below=3, no\_above=1)

d.add\_documents([[UNK, PAD]])

d.compactify()

**return** d

This code is used to create the dictionary that was mentioned in the methodology section. This is also used to convert the raw text input into a list of tokens that will be used as input to the neural network.  
x\_train = texts\_to\_input(df.review, dictionary) x\_train

Converts df.reviews into list of tokens and stores it in x\_train

After the dataset is processed. x\_train, y\_train, x\_test, y\_test are created for training and testing purposes. y\_train and y\_test are created by converting the labels ‘positive’, ‘negative’ into 1 and 0.

df['sentiment'] = df['sentiment'].map({'pos': 1, 'neg': 0})

Next, we move on to building the model. The model is built carefully by choosing the right parameters and layers. This is done to ensure maximum efficiency and minimum loss. The model is trained over 3 epochs.

model = Sequential()

model.add(Embedding(len(dictionary), embed\_size))

model.add(Bidirectional(LSTM(32, return\_sequences = **True**)))

model.add(GlobalMaxPool1D())

model.add(Dense(20, activation="relu"))

model.add(Dropout(0.05))

model.add(Dense(1, activation="sigmoid"))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

batch\_size = 100

epochs = 3

model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_split=0.2)

The output looks like this.

Train on 60000 samples, validate on 15000 samples

Epoch 1/3

60000/60000 [==============================] - 271s 5ms/sample - loss: 0.3670 - acc: 0.8293 - val\_loss: 0.2173 - val\_acc: 0.9251

Epoch 2/3

60000/60000 [==============================] - 271s 5ms/sample - loss: 0.1643 - acc: 0.9402 - val\_loss: 0.1007 - val\_acc: 0.9669

Epoch 3/3

60000/60000 [==============================] - 268s 4ms/sample - loss: 0.0728 - acc: 0.9759 - val\_loss: 0.0648 - val\_acc: 0.9782

The other 2 models are quite similar, only the embed\_size and the dictionary size changes.

Once the model is trained, it is saved using

model.save('sentiment\_model-v2.h5')

The test set is used to make predictions and evaluate the model. It looks something like this.

prediction = model.predict(x\_test)

y\_pred = (prediction > 0.5)

**from** **sklearn.metrics** **import** f1\_score

print('F1-score: **{0}**'.format(f1\_score(y\_pred, y\_test)))

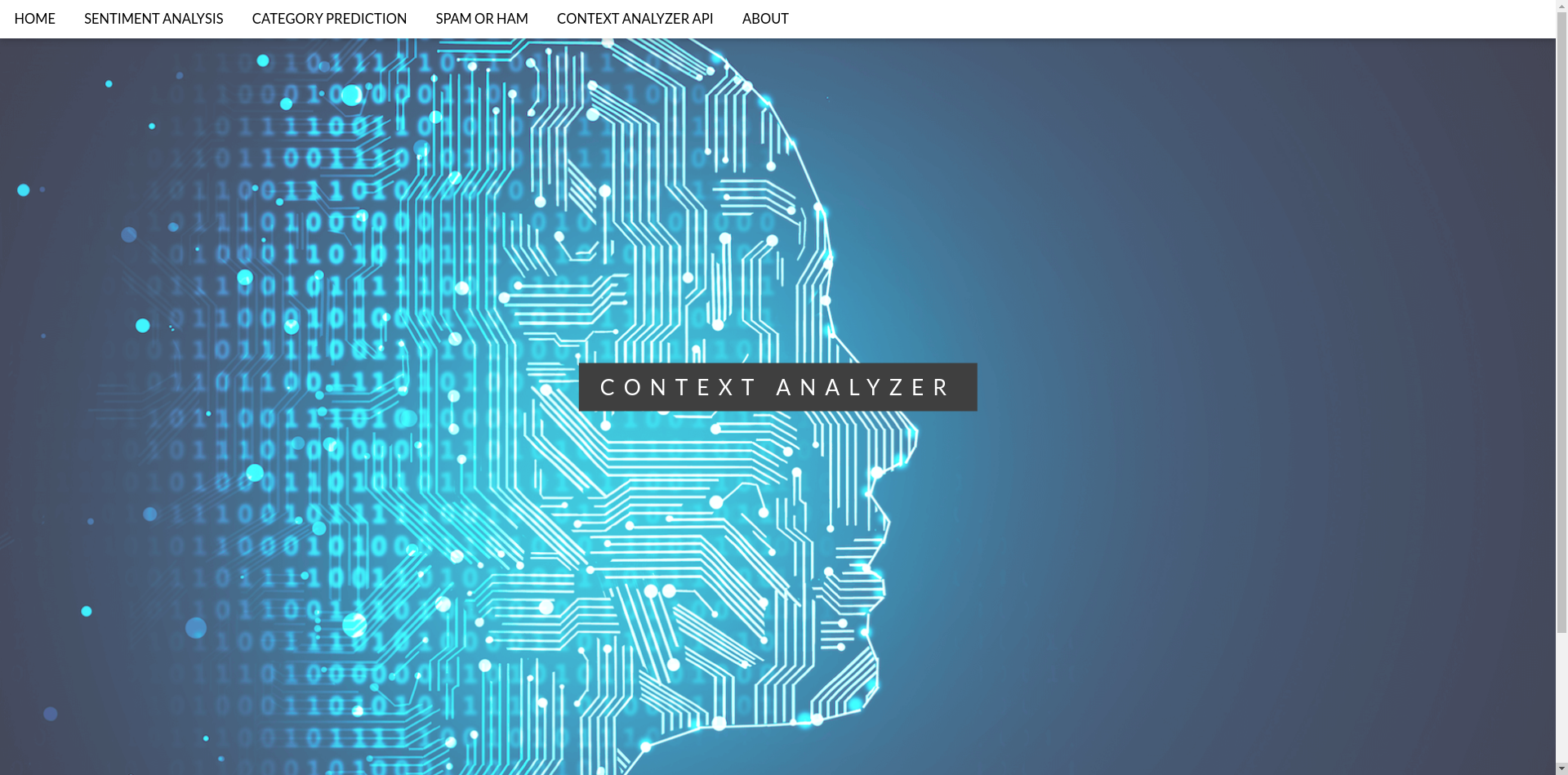
F1-score: 0.9861099959855479

This covers the entire project in brief. However there is still one tiny, yet important part to be done. The tensorflow.js code can not import .h5 files. It can only import .json models. So the model.h5 files have to be converted into model.json files. This is done easily in just one line.

$ tensorflowjs\_converter --input\_format=keras /tmp/model.h5 /tmp/tfjs\_model

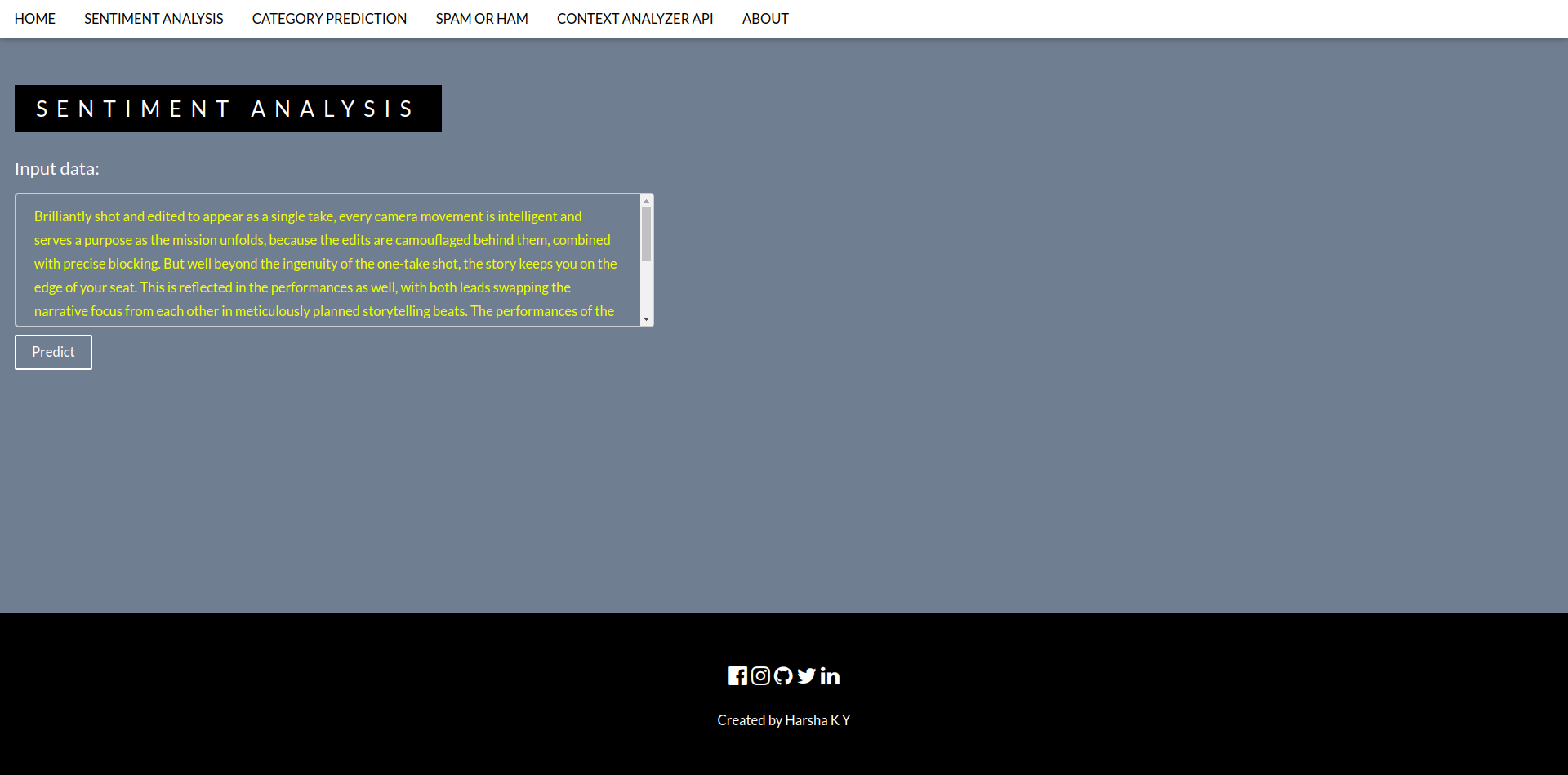
**6.2. Screenshots**

1. Home



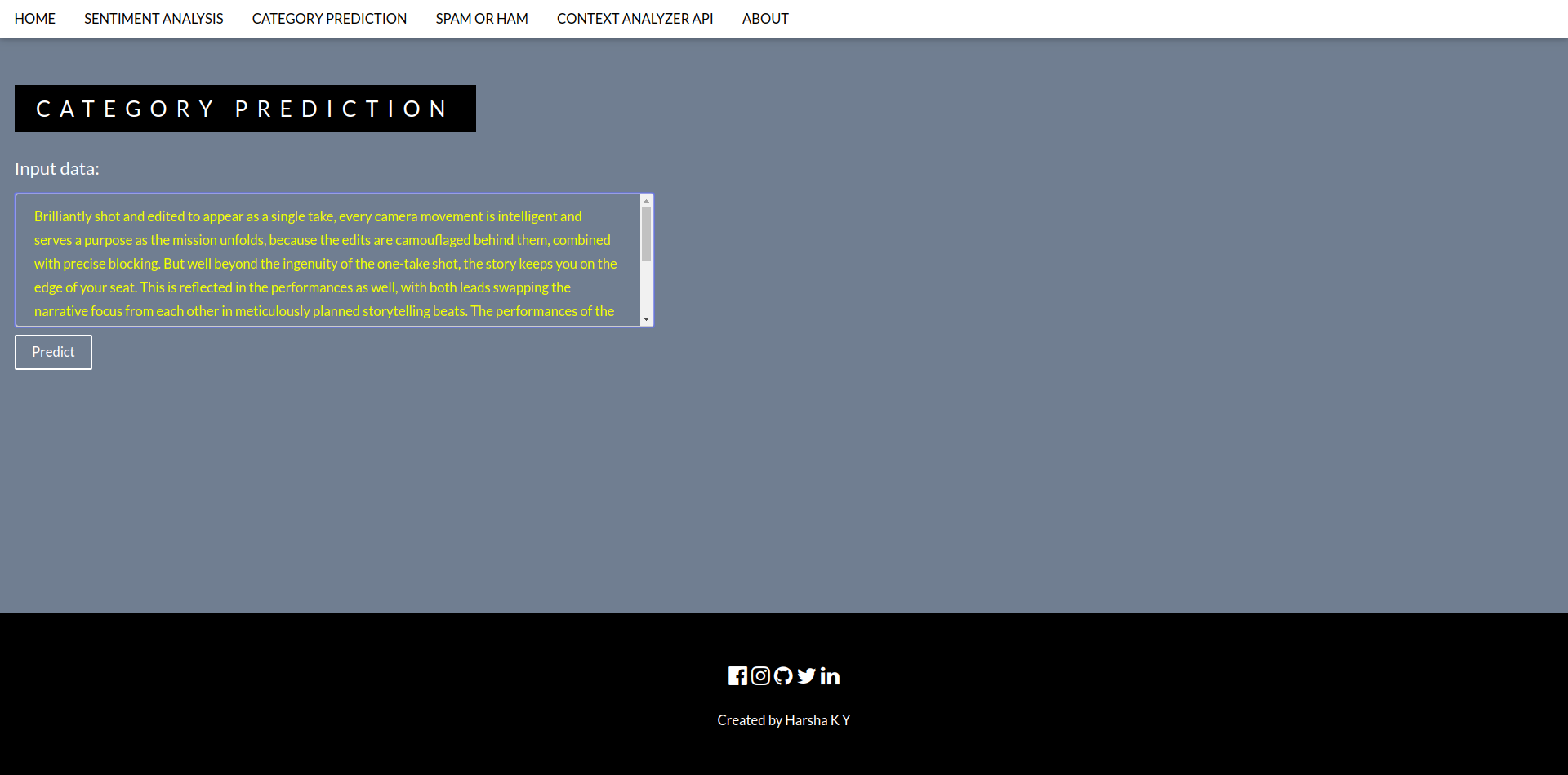
**Figure 7.1.a – Home Page**

1. Sentiment Analysis



**Figure 7.1.b – Sentiment Analysis**

1. Category Prediction



1. Spam Detection
2. Context Anayzer API (Documentation)
3. JSON responses

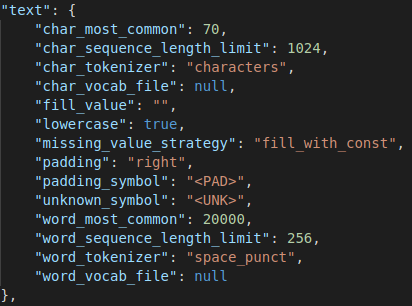
# 8. RESULT AND DISCUSSION

After the model is created, it is deployed as a production API where the APIs can be used for real-time predictions. It accepts one row of data and provides a predicted result based on your model for that data. You can use predictions when you need a prediction as input for your business logic flow.

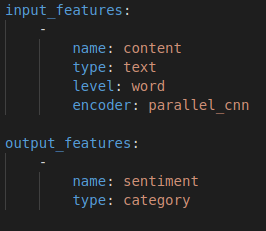
## 8.1 Evaluate JSON and Feature Importance

It tells you what features it considers to be most important for building this model in the Chart of Value. The function value is calculated by measuring the impact that each function has on the prediction when it is disturbed over a wide range of values sampled from the dataset. You must review this information to ensure that all of the most important features make sense of your data. Micro-averaged accuracy is determined by adding together the number of true positive (TP) for each potential value of the target column and dividing it by the number of true positive (TP) and true negatives (TN) for each potential value. A score threshold is a number which ranges from 0 to 1. It provides a way to determine the minimum level of confidence where the value of the prediction should be taken as true.

For example, if you have a class that is unlikely to be the actual value, you would want to lower the threshold for that class; using a threshold of 5 or higher would make that class extremely rarely (or never) expected. The higher threshold reduces false positives, at the cost of more false negatives. The lower threshold reduces false negatives at the cost of more false positives. Put another way, the score threshold affects accuracy and recall. Higher threshold increase accuracy (because the model never predicts unless it is extremely certain) but decreases recall (the percentage of positive examples that the model is correct). Below (Figure 8.1) shows the importance of a function for the sample dataset selected. And Figure 8.2 indicates the features selected by the customer.



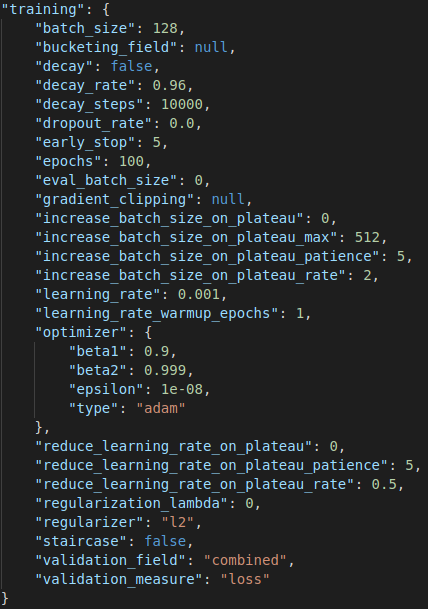
## Figure: 8.1: Text Feature Importance



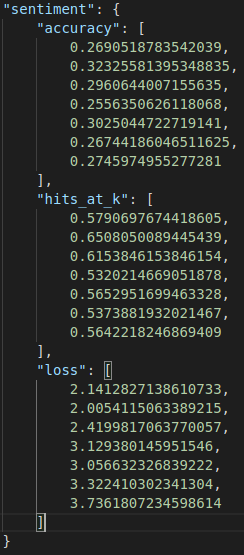
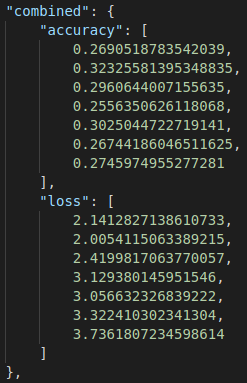
## Figure: 8.2: Text Feature Importance

Creating many more features and engineering by combining them is just the first step in building a good ML model. If we try to generate a set of new features based on just a few technical rules and finitely combine them, it will quickly lead to a combinatorial explosion of possible features to be used. Moreover, most of the new features are not very useful. We need a strategy to eliminate unnecessary features and work with just a subset that can have the most impact on our ML models. Fortunately, many of these strategies can be automated and implemented in a systematic programmatic manner.

Here's an example of a training course:

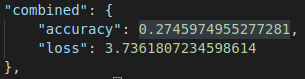


## Figure: 8.3: Training Parameters



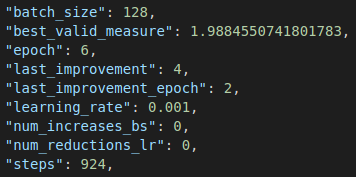
## Figure: 8.4: Accuracy and Loss – Combined and Separated

Selecting features based on the value of some test model. We build a model, not necessarily the best one for a given problem, and then look at the value of that model. For example, we can look at the absolute values of the linear regression coefficients provided by the tree-based model. We may set some threshold and delete all the functions that fall below it. Forward collection and/or recursive deletion of features. In this method, we add one-by-one features to the model and retain only those for which the model improves or remove functions and keep only those for which the model deteriorates. Permutation effect: we change the values of the features, one by one. A function is important if the shuffling (permuting) values increase the predictive error and are unimportant if the shuffling does not have an impact on the prediction error.

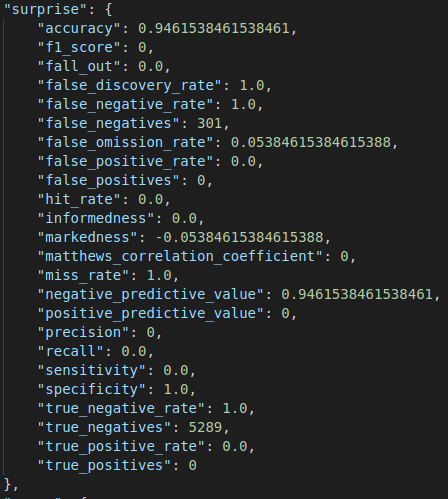


## Figure: 8.5: Test Accuracy and Loss

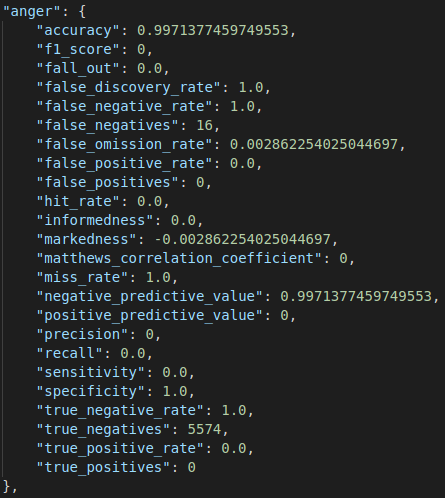
Function collection of genetic algorithms. Genetic algorithms are based on their biological counterparts. This approach creates a lot of randomly shuffled features and tests their "fitness" based on how well they support the ML algorithm. The process is repeated several times, each equivalent to a step in natural evolutionary selection. After many of these measures, only the "fittest" genes (sets of features) can "survive."



## Figure: 8.6: Test parameters and final model parameters



## Figure: 8.7: Sample sentiment parameters - *Surprise*



## Figure: 8.8: Sample sentiment parameters - *Anger*

# 9. Testing

## 9.1. Continuous Integration

Agile and DevOps continue to spread to IT ventures at a rate we've never seen before. As interest in these issues increases, there is also an increase in the set of processes and tools that make it possible to deliver better applications. The "Phase" execution is at the end of the software development lifecycle. And if delivery is not swift, the entire Agile development cycle may be disrupted. Modern software should be designed to be distributed easily, using suitable automation tools. Cloud Build is a fully managed Google Cloud Platform service that lets you build software quickly across all languages, relying on Docker to get the job done. Code testing is a way to check the system to differentiate any blunders, gaps or missing pre-requisites from the real necessity. Programming testing is comprehensively divided into two types-utilitarian testing and non-useful testing. At the point of starting the test exercises: testing should be started as timely as it is possible to reduce the cost and time to rewrite and build code that is bug-free so that it continues to be passed on to the consumer. However, in the Software Development Life Cycle (SDLC), testing can be started from the Requirements Gathering stage and continued until the product is out in the preparations. It also relies on the model of advancement that is being used. For example, in the Waterfall model, testing begins at the test stage which is very low in the tree; but in the V-model, testing is carried out parallel to the progress stage.

## 9.2. Unit Testing

Unit Testing is the programming evaluation aspect where the individual units/parts of the software are evaluated. The reason for the existence is to allow the execution of each unit of the product as anticipated. The unit is the smallest testable piece of any object. More often than not, it has one or a few sources of information and, as a consequence, a single yield. Through procedural programming, the unit could be an individual program, function, process, and so on. The smallest unit in the programmed element is a tactic that may have a position with a base/superclass, a special class or a determined / youngster class. (Some view the application module as a product.

This is to be undermined as there will most likely be a large number of individual units inside the module). Unit testing devices, drivers, stubs, and counterfeit/counterfeit products are used for unit testing.

### 9.2.1. Unit Testing Benefits:

Device testing builds confidence in modifying / searching for code. On the one possibility the major device tests will be written and on the off chance that they will be run any time the code is changed, we will most likely automatically get to know any imperfections due to the change. Likewise, because, as of now, codes are less associated to allow unit testing possible, the unintended impact of any code changes is less. Codes are becoming rapidly reusable. Codes should be omitted to make unit testing potential. This means that codes are easier to reuse. Production is quicker than that. On the chance that you don't have a unit testing setup, you're composing your code; (you set up some breakpoints, fire up the GUI, send a few data sources that hopefully hit your code and expect you to be on the right track.) But, if you have a unit testing set up, you're writing the test, composing the code, and running the test. Composing the tests requires a substantial investment, however, the time is remunerated by the less time it takes to run the tests; you do not need to start the GUI and have each of these sources of information. What more unit tests are more reliable than model tests. Advancement is also faster over the long haul. The energy needed to discover and repair the dropouts discovered in the mid-unit test is less in contrast to the effort required to correct the absconds discovered in the device test or recognition test. The cost of correcting an imperfection detected in the unit check is lower relative to the cost of deformities reported in higher amounts. Look at the expense (time, effort, obliteration, embarrassment) of an imperfection that has been identified in the recognition test or when the product is alive.

### 9.2.2. Unit Test Cases

### Table 9.1 shows the list of unit test cases that will be run by continuous integration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Test Case** | **Scenario** | **Input** | **Expected Result** | **Actual Result** | **Result** |
| **TC01** | **POST** /users | Checking authorization | Email ID | Log In | Logged In | PASS |
| **TC02** | **POST** /users | Checking authorization | Wrong Email ID | Error | Error: Wrong Email | PASS |
| **TC03** | **POST** /users  /sign\_in | Authentication | AdminToken | Unique ID | Unique IDgenerated successfuly | PASS |
| **TC04** | **POST** /users  /sign\_in | Authentication | Different AdminToken | Duplicate  ID | Error: Wrong auth token | PASS |
| **TC05** | **GET** /datasets | Retrieve stored user data | GET-request | Stored datasets | Stored datasets are fetched | PASS |
| **TC06** | **GET** /datasets | Retrieve stored user data | Some other request | Error | Error: Improper request | PASS |
| **TC07** | User Selects Dataset | Selection | Data | Upload  Data | Dataset uploaded succesfully | PASS |
| **TC08** | User Selects Dataset | Selection | Data with imporoper featues | Upload  Data | Dataset uploaded succesfully | PASS |
| **TC09** | User Selects Features | Feature Engineering | Data | Selected Features | Proceed for ML | PASS |
| **TC09** | User Selects Features | Feature Engineering | Data with improper features | Improper Features | Do not proceed for ML | PASS |
| **TC10** | User users prediction APIs | Prediction | Row(s) of Data to be predicted | Prediction | Output of predicted values | PASS |
| **TC11** | User users prediction APIs | Prediction | Row(s) of improper Data to be predicted | Error | Improper: data passed | PASS |
| **TC12** | User users prediction APIs | Prediction | Row(s) of Data something lese | Error | Error | PASS |

## 9.3. Model Testing

To compare the two methods, the manual method and the project, we used a test problem indicative of a real-world use case for sentiment analysis: we used a text dataset with a categorical result of sentiment. The advantages of an accurate model are evident, as they allow the sentiment analysis to quickly identify cases of emotions in sentences.

### 9.3.1. Trying out this application

Started with this project, using their deep learning module to build a model automatically. Importing data and setting up a job was easy, although the tool has little flexibility on how to arrange and mark import data. Below is a screenshot of the results page that shows the results after < 15 minutes of training. AutoML has produced an impressive 81.3 per cent accuracy and 78.9 per cent recall on this task, without requiring a single line of code! It also provides an API to access the model we generated for use in the generation of predictions.

### 9.3.2. Using a Hand-Crafted Model with TensorFlow

Create our model from scratch using TensorFlow and TensorFlow Extended Library, which allows it easier to create and train models in TensorFlow. We used data increase on our training dataset by using vision transforms to construct a more reliable model, tuning the hyperparameters as well as downscaling and then upscaling the images in training iterations. With a few hours of work and ~15 minutes of training time, we obtained an accuracy of 82.7 per cent and a recall of 81.0 per cent, exceeding Google AutoML tests by 1–2 per cent.

### 9.3.3. Testing Results

While we have obtained superior results on this test problem through a handcrafted neural network over this process, it has also been significantly more time-consuming and complex. Even with the aid of TensorFlow Extended Library and TensorFlow, it took ~2 hours to set up and train our product. Once we were set up with AutoML, it took no more than 30 minutes to import our data and train the model without writing a single line of code. As far as usability and ease of use are concerned, AutoML wins hands down. However, when your dataset is not properly organized into named folders or you want to use custom validation sets, things get complicated. Today, AutoML is relatively linear in its interface and greatly reduces the flexibility and distinction that can be accomplished.

It fits well with "textbook" issues, but for many real-world problems with added complexity in inputs, names, or validation techniques, it just can't handle it today. Although we have achieved better precision modeling by hand, the significance of the 1–2 percent increase in accuracy we have achieved depends on the specific problem being resolved. In some cases, this may not be a significant difference, and in other cases, it may make or damage the value of the model. Overall, AutoML is a fine, easy-to-use tool to quickly build "good enough" models for standard classification tasks with no coding required. However, in its current form, it quickly breaks down problems with complicated data structure, custom labelling approaches or custom validation techniques often used in practice due to its applicability to real-world machine learning. It also serves as a black-box model, not revealing any of the internal intestines for inspection and subsequent adjustment as required for further training or development. In our experience, black-box models are almost always a bad idea when it comes to making predictions in the real world.

# 10. Conclusion

It's amazing to see the tremendous progress that has been made in automating deep learning over the last few years. It makes it more available to consumers and businesses; the potential of deep learning is made more accessible to the general public. But there's always room for improvement. Design search has become much more efficient; finding a single GPU network in a single training day is pretty amazing. However, our quest space is still very limited. Existing algorithms also use hand-designed constructs and building blocks, but tie them together differently! A successful and potentially ground-breaking future direction would be a far wider search for new architectures. These algorithms will reveal even more hidden deep learning secrets inside these large and complex networks. Of course, such a search space requires the efficient design of an algorithm. This new project is creating exciting challenges for the AI community and a chance for another advance in science. Overall, there are several options for using this project today. It just depends on whether you're going to play around with the algorithm you want and how much you're willing to pay to get some more code out of it. We live in an era where data growth outpaces our ability to make sense of it. This is explained not only by the current technological obstacles but also by our reliance on experts to carry out this mission. AutoML is an exciting field that has been on the spotlight and promises to alleviate this problem by intelligent automation of repetitive ML workflow tasks. Machine learning has become one of the main engines of the present era. The production line of machine learning models passes through different phases and stages, requiring a wide knowledge of several available tools and algorithms. However, as the amount of data generated daily continues to increase on an exponential scale, it has become important to automate this process. In this study, the state-of-the-art research effort in the field of AutoML frameworks has been comprehensively covered. We also highlighted research directions and open complexities that need to be addressed to achieve the vision and objectives of the AutoML project. We hope that our survey will serve as a useful resource for the community, both researchers and practitioners, to understand the challenges of the field and provide useful insights to further advance the state-of-the-art in several directions.

# 11. Future Enhancement

Essentially, the purpose of this project is to automate repetitive tasks such as pipeline development and hyperparameter tuning so that data scientists will focus more of their time on the business issue at hand. It also aims to make technology available to everyone rather than to a select few. This technology and data scientists can operate together to improve the ML cycle so that the true usefulness of machine learning can be used. Whether or not this project becomes a success depends mainly on its implementation and the progress made in this field. Nevertheless, the application is a big part of the future of machine learning. It will also handle most of the data-cleaning process It will greatly improve deep learning It will be applied to large data sets It will become human competitive It will change the nature of data science as we know this It is only a small part of a greater meta-learning revolution If you are employed in data science today or managing a team of data scientists, you will interpret and implement it as follows. Although more research efforts have been made in recent years to address the complexities of automated machine learning, there is still a range of open challenges and research directions that need to be tackled to achieve the ultimate goals and vision of the AutoML domain. In this segment, we highlight some of the issues that need to be tackled to improve the state of the art. Scalability: In reality, the main limitation of centralized systems for automating approaches to the CASH problem (e.g. Auto-Weka, Auto-Sklearn) is that they are closely linked to a machine learning library (e.g. Weka, sci-kit-learn, R) that can only work on a single node that makes them not applicable to large data volumes. In practice, as the amount of data generated daily continues to increase on an exponential scale, many distributed machine learning systems have recently been introduced.

Although some initial efforts have been made to develop a distributed automated framework for the CASH issue. However, the proposed distributed solutions are still simple and limited in their ability. More research efforts and new solutions are needed to address the challenge of automatically constructing and tuning machine learning models over massive datasets. Optimization Techniques: In practice, different AutoML frameworks use different hyperparameter optimization techniques for machine learning algorithms.

For example, Auto-Weka and Auto-Sklearn use the SMAC cross-validation technique during hyper-parameter configuration optimization and assessment. TPOT uses genetic programming and Pareto optimization to create candidate pipelines. It is difficult, in practice, to find a clear winner or one-size-fits-all technique. In other words, there is no single approach that can outperform all other techniques on different datasets with their different characteristics, types of search spaces and metrics (e.g. time and accuracy). There is, therefore, a crucial need to understand the pros and cons of these optimization techniques so that AutoML systems can automatically tune their hyperparameter optimization techniques or their strategy for exploring and traversing the search space. Such decision automating should provide improved performance over the selection process and depend on a defined strategy. Similarly, for the various meta-learning techniques adopted, there is no clear systematic method or evaluation metric to quantitatively evaluate and compare the effect of these techniques on the reduction of the search space.

## 11.1. User-friendliness

In general, most of the software and system currently in use cannot be called user-friendly. We still need advanced technical expertise to be applied and used. Such a challenge restricts its usefulness and wide acceptance among lay users and domain experts (e.g. doctors, accountants) who often have limited technical skills. One of the solutions to overcoming these problems may be the provision of an accessible and lightweight web interface for such a framework. Continuous delivery pipeline: continuous delivery is characterized as the development of a repeatable, consistent and incremental improvement mechanism for the transition of software from concept to the customer. Integrating machine learning models into continuous delivery pipelines for successful use has not gained much attention recently, as data scientists typically move them straight into the production environment with all the disadvantages that this method may have, such as no proper unit testing and integration testing. Data Validation: In this sense, most approaches in the literature concentrate on issue identification and user notification only. Nonetheless, an automated correction has not been studied in a good way, covering many potential domains of datasets and that the role of a data scientist in machine learning production.

Also, as possible data repair is an NP-Hard issue, there is a need to find more approximation techniques to solve this problem.

## 11.2. Data Preparation

There is a critical need to optimize the extraction process as it is known to be one of the most time-consuming sections of the pipeline. Throughout reality, most systems neglect the automation of moving data features to different domain spaces, such as performing main component analysis, or linear discriminant analysis, and improving model performance. Even, we assume that there is room for improvement of existing types of auto-encoders, such as Boltzmann Restricted Machines. Further research is needed, therefore, to try out different architectures and interpret them to be able to automate the choice of suitable encoders. Also, there are different methods for determining the value of a ranking, which is a very important part of the process of automating the selection process. However, there are no systematic comparative studies between these methods or effective recipes that can be recommended when using each of these techniques.

## 11.3. Model Deployment and Life Cycle

Recently, some tools and mechanisms have been put in place to make data scientist work easier and to automate machine learning development. In reality, however, there is still a need to incorporate these different systems along the entire pipeline. There is sufficient room for improvement, for example, about the automatic choice of good workflows appropriate to each problem and how to integrate more data comprehension, testing and planning techniques with workflows. For particular, these frameworks still do not provide the end-user with any smartness in the decision-making process that is the cornerstone of eliminating the role of a human being in the loop.

# Appendix A: Bibliography

# Appendix B: User Manual

1. Click on **Sign Up** and create an account for yourself. You can choose your appropriate **Email ID** and **Password.**
2. After the registration, you will be redirected to the **Main Page** of the application which is the **Data Set List Page**
3. Click on **Upload Dataset** and upload the **CSV** file to which you want to apply **Machine Learning** to.
4. Proceed next to Feature Engineering. Here, select the Input Variables that you want to predict from and Output Variables that need to be predicted
5. Wait for around 15 mins (depends on the dataset) and then you will see results along with the URL for predictions API
6. Send a POST request to the URL with your features for prediction. The URL will return a prediction JSON along with the parameters and related information.